HTL St. Johann Informatik

Netzwerkprogrammierung

Inhalte Netzwerkprogrammierung

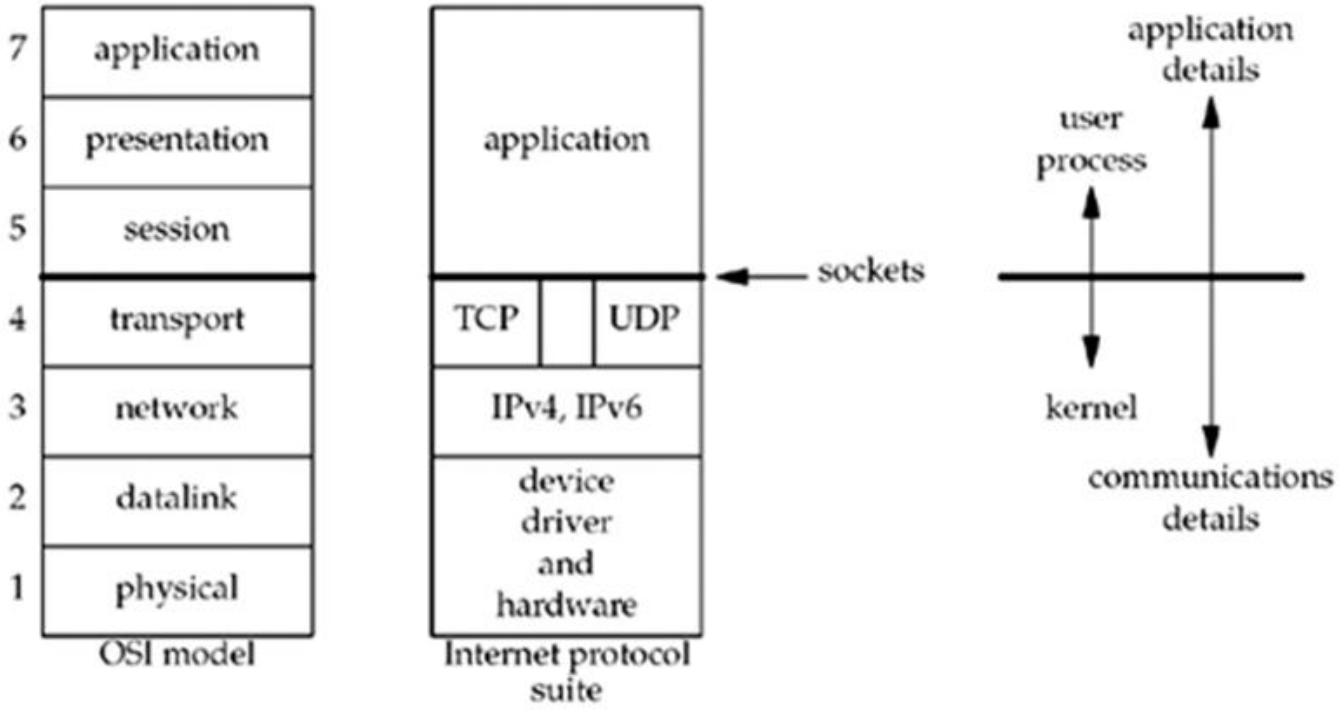
1. Netzwerkprogrammierung mit Sockets
2. TCP und UDP Verbindungen
3. Konfiguration Client - Server
4. Codebeispiele (C, Python, Java)
5. Zusammenfassung

Netzwerkprogrammierung

- Teil der Interprozesskommunikation - von *Personal Computer* zu *Netzrechner*
- Anwendungen: IoT, Verteilte Systeme, M2M Protokoll, …
- Implementierung durch *Sockets*

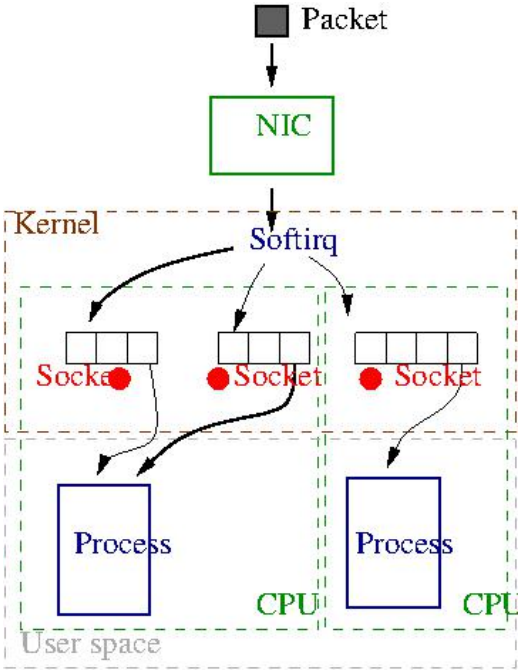
	Gemeinsamer Speicher	Nachrichten-warteschlangen	Pipes	Sockets
Art der Kommunikation	Speicherbasiert	Objektbasiert	Datenstrombasiert	Nachrichtenbasiert
Bidirektional	ja	nein	bei benannten Pipes	ja
Plattformunabhäng	nein	nein	nein	ja
Prozesse müssen verwandt sein	nein	nein	bei anonymen Pipes	nein
Kommunikation über Rechnergrenzen	nein	nein	nein	ja
Bleiben ohne gebundenen Prozess erhalten	ja	ja	nein	nein
Automatische Synchronisierung	nein	ja	ja	ja

1 Sockets als Schnittstelle



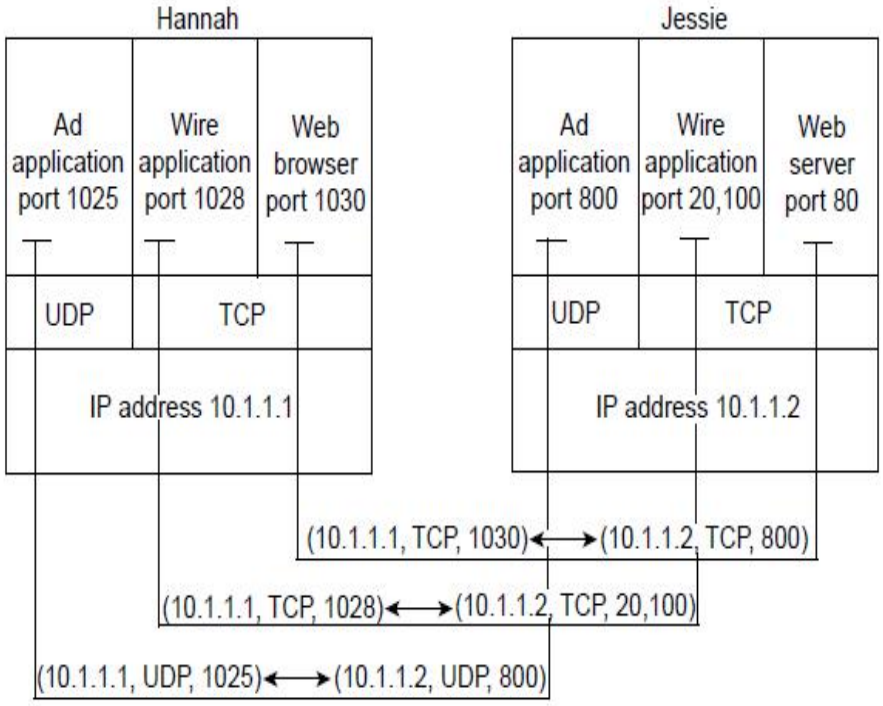
Funktionsweise Sockets

- **Endpunkte** für bidirektionale Interprozesskommunikation in verteilten Systemen**,** in jeder Programmiersprache verfügbar
- Benutzerprozess kann **Socket vom OS anfordern**, über diesen Daten senden und empfangen
- **OS verwaltet alle benutzten Sockets** und zugehörige Verbindungsinformationen



Adressierung mit Sockets

- Daten basierend auf mit *file descriptor (fd)* gelesen und geschrieben
- Konfiguration eines Socket: IP, Portnummer, TCP / UDP



2 TCP und UDP Verbindungen

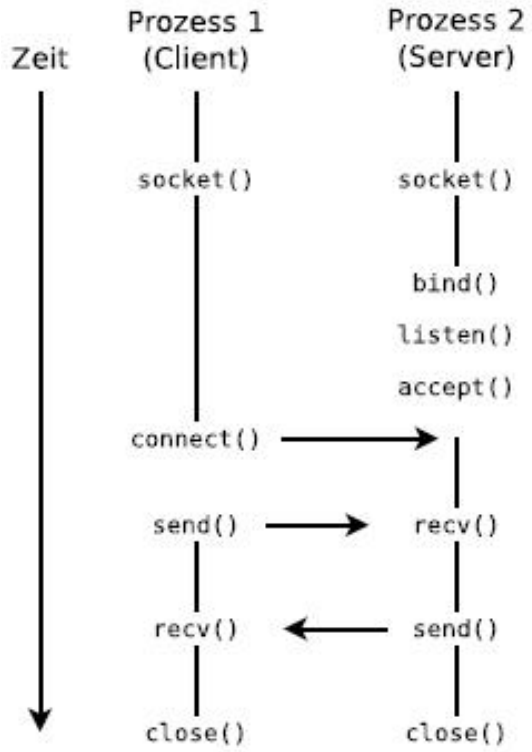
- **Verbindungsorientierte Sockets** (***Stream Sockets***)
  - Verwendet Transportprotokoll TCP
  - *Vorteil*: Höhere Verlässlichkeit, Segmente können nicht verloren gehen und kommen immer in der korrekten Reihenfolge an
  - *Nachteil*: Geringere Geschwindigkeit als bei UDP

- **Verbindungslose Sockets** (***Datagram Sockets***)
  - Verwenden das Transportprotokoll UDP
  - *Vorteil*: Höhere Geschwindigkeit als bei TCP, da geringer Aufwand (Overhead) für das Protokoll
  - *Nachteil*: Segmente können einander überholen oder verloren gehen

TCP Socket

- **Client**
  1. Socket erstellen (*socket*)
  2. Client mit Server-Socket verbinden (*connect*)
  3. Daten senden (*send*) und empfangen (*recv*)
  4. Socket schließen (*close*)

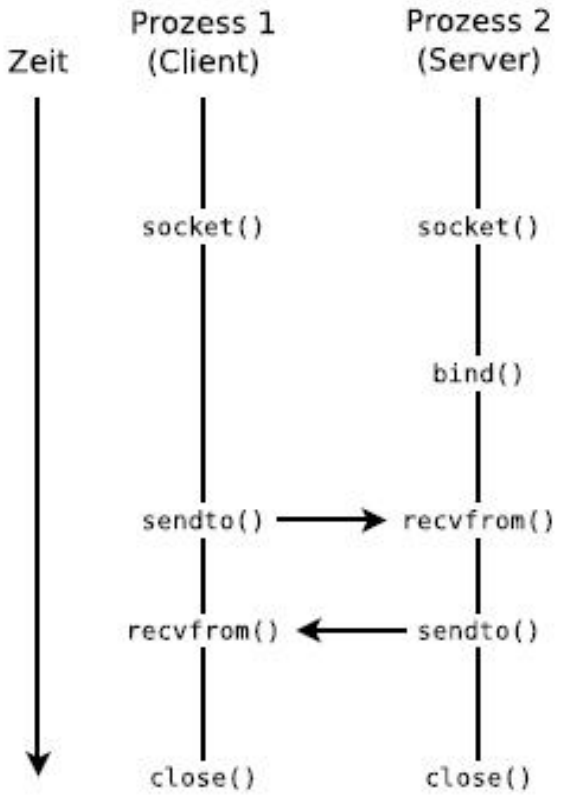
- **Server**
  1. Socket erstellen (*socket*)
  2. Port an einen Socket binden (*bind*)
  3. Socket empfangsbereit machen (*listen*) und Warteschlange für Clients einrichten
  4. Verbindungsanforderung (*accept*) akzeptieren
  5. Daten senden (*send*) und empfangen (*recv*)
  6. Socket schließen (*close*)



UDP Socket

- **Client**
  1. Socket erstellen (*socket*)
  2. Daten senden (*sendto*) und empfangen (*recvfrom*)
  3. Socket schließen (*close*)

- **Server**
  1. Socket erstellen (*socket*)
  2. Port an Socket binden (*bind*)
  3. Daten senden (*sendto*) und empfangen (*recvfrom*)
  4. Socket schließen (*close*)



3 Konfiguration TCP Client

```
# include <sys/types.h>
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>

int sock_fd;
struct sockaddr_in server_addr;

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(80);
inet_aton("127.0.0.1", &(server_addr.sin_addr));

err = connect(sock_fd, (struct sockaddr *)&server_addr,
              sizeof(struct sockaddr_in));
if (err == -1)
  perror("connect() failed");
```

Filedeskriptor für Datenaustausch

Struktur für **Serveradressierung**

Konfiguration **Serveradressierung**

Verbindungsaufbau zu Server

Datenaustausch TCP Client

```
while (1)
 {
  FD_ZERO(&input_fdset);
  FD_SET(STDIN_FILENO, &input_fdset);
  FD_SET(sock_fd, &input_fdset);
  if (select(sock_fd+1, &input_fdset, NULL, NULL, NULL)
      == -1)
    perror("connect: select() failed");
  if (FD_ISSET(STDIN_FILENO, &input_fdset))
   {
    if (fgets(buffer, 256, stdin) == NULL)
     {
      printf("connect: Closing socket.\n");
      break;
     }
    length = strlen(buffer);
    send(sock_fd, buffer, length, 0);
   }
  else
   {
    length = recv(sock_fd, buffer, 256, 0);
    if (length == 0)
     {

      printf("Connection closed by remote host.\n");
      break;
     }
    write(STDOUT_FILENO, buffer, length);
   }
 }
```

Verbindung in Endlosschleife

Polling des *fd*, ob Daten Vorhanden sind

Dateneingabe von Client

Gemeinsamer Puffer zum Lesen und Schreiben (IPC)

Datenempfang von Server

Konfiguration TCP Server

```
                                /*--- socket() ---*/
sock_fd = socket(PF_INET, SOCK_STREAM, 0);
if (sock_fd == -1)
  err_exit("server: Can't create new socket");

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(port);
my_addr.sin_addr.s_addr = INADDR_ANY;
```

Socket Konfiguration

```
                                /*--- bind() ---*/
err = bind(sock_fd, (struct sockaddr *)&my_addr,
           sizeof(struct sockaddr_in));
if (err == -1)
  err_exit("server: bind() failed");
```

Port an Socket binden

```
                                /*--- listen() ---*/
err = listen(sock_fd, 1);
if (err == -1)
  err_exit("server: listen() failed");
```

Auf Anbindungen warten,
Client Warteschlange auf 1 begrenzt

```
                                /*--- accept() ---*/
addr_size = sizeof(struct sockaddr_in);
client_fd = accept(sock_fd,
        (struct sockaddr *)&client_addr, &addr_size);
if (client_fd == -1)
  err_exit("server: accept() failed");
printf("I'm connected from %s\n",
       inet_ntoa(client_addr.sin_addr));
```

Speicherung von IP und Port des Clients

Datenaustausch TCP Server

```
if (FD_ISSET(STDIN_FILENO, &input_fdset))
 {
  if (fgets(buffer, 256, stdin) == NULL)
   {
    printf("server: Closing socket.\n");
    break;
   }
  length = strlen(buffer);
  send(client_fd, buffer, length, 0);
 }
else
 {
  length = recv(client_fd, buffer, 256, 0);
  if (length == 0)
   {
    printf("Connection closed by remote host.\n");
    break;
   }
  write(STDOUT_FILENO, buffer, length);
 }
}
close(client_fd);
close(sock_fd);
```

Daten für Client erfassen

Daten zum Client senden

Daten vom Client aus Puffer lesen

Daten vom Client ausgeben

Client- und Serversockets
Bidirektional schließen

4 Python UDP Server

udpServerSimple.py > receive_udp_data

```python
import socket

def receive_udp_data():
    # IP-Adresse und Portnummer
    udp_ip = "localhost"
    udp_port = 5000

    # UDP-Socket erstellen und binden
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((udp_ip, udp_port))

    print(f"Warte auf Daten auf {udp_ip}:{udp_port}...")

    try:
        while True:
            # Daten empfangen
            data, addr = sock.recvfrom(1024)
            print (f"From: {addr}: {data.decode('utf-8')}")

    except KeyboardInterrupt:
        print("Programm beendet.")
    finally:
        sock.close()

if __name__ == "__main__":
    receive_udp_data()

```

4 Python UDP Client

udpClientSimple.py > ...

```python
import socket

def send_udp_data():
    # IP-Adresse und Portnummer
    udp_ip = "localhost"
    udp_port = 5000

    # UDP-Socket erstellen
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # bind ist optional. Wird es nicht aufgerufen, weißt das Betriebssystem selbst einen
    # frein Port zu.
    # Bitte ausprobieren
    sock.bind((udp_ip, 50001))
    try:
        while True:
            txt = input()

            # Daten senden
            sock.sendto(txt.encode('utf-8'), (udp_ip, udp_port))


    except KeyboardInterrupt:
        print("Programm beendet.")
    finally:
        sock.close()

if __name__ == "__main__":
    send_udp_data()
```

4 Python TCP Client

```
4
5  import socket                        # Modul socket importieren
6
7  HOST = 'localhost'                   # Hostname von Server
8  PORT = 50007                         # Portnummer von Server
9
10 # Socket erzeugen und Socket Deskriptor zurückliefern
11 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 sd.connect(HOST, PORT)              # Mit Server-Socket verbinden
14
15 sd.send('Hello, world')            # Daten senden
16
17 data = sd.recv(1024)               # Daten empfangen
18
19 sd.close()                         # Socket schließen
20
21 print 'Empfangen:', repr(data) # Empfangene Daten ausgeben
```

Python TCP Server

```
4
5  import socket                # Modul socket importieren
6
7  HOST = ''                    # '' = alle Schnittstellen
8  PORT = 50007                 # Portnummer von Server
9
10 # Socket erzeugen und Socket Deskriptor zurückliefern
11 sd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12
13 sd.bind(HOST, PORT)          # Socket an Port binden
14
15 sd.listen(1)                 # Socket empfangsbereit machen
16                              # Max. Anzahl Verbindungen = 1
17
18 conn, addr = sd.accept()     # Socket akzeptiert Verbindungen
19
20 print 'Connected by', addr
21 while 1:                     # Endlosschleife
22     data = conn.recv(1024)   # Daten empfangen
23     if not data: break       # Endlosschleife abbrechen
24     conn.send(data)          # Empfangene Daten zurücksenden
25
26 conn.close()                 # Socket schließen
```

Java Client

```
package netzwerk;
import java.io.*;
import java.net.*;

public class Client {

    static void init() throws IOException {
      Socket server = new Socket ("10.0.80.55", 3141);
      InputStream in = server.getInputStream();
      OutputStream out = server.getOutputStream();

      out.write(4);
      out.write(2);
      int result = in.read();
      System.out.println(result);
      server.close();
    }

  public static void main(String[] args) {

  try{
    init();
    }
    catch (IOException e) {}
  }
}
```

Java Server

```
package netzwerk;
import java.io.*;
import java.net.*;

public class Server{
private static void handleConnection (Socket client) throws IOException{
        InputStream in=  client.getInputStream();
        OutputStream out = client.getOutputStream();

        int factor1=in.read();
        int factor2=in.read();
        out.write(factor1 * factor2);
System.out.println(factor1 * factor2);
}

public static void main (String [] args) throws IOException{
        ServerSocket server= new ServerSocket (3141);
                while(true){
                Socket client=null;

                try{
                client=server.accept();
                handleConnection (client);
                }

catch (IOException e){
e.printStackTrace();
}
finally {
if (client !=null)
try {client.close();} catch (IOException e) {}
}}
}}
```

Zusammenfassung

- **Motivation**
  - Netzwerkprogrammierung erweitert Rechnergrenzen
  - Stabile und getestete Systemfunktionen vorhanden
  - Systemnahe bietet hohe Sicherheit

- **Implementierung**
  - Sockets als Endpunkte für bidirektionale Kommunikation
  - Systemfunktionen für TCP/UDP Datenkommunikation
  - Viele Libaries für Erweiterungen vorhanden